A Parametric Abstract Domain for Lattice-Valued Regular Expressions

Midtgaard, Jan; Nielson, Flemming; Nielson, Hanne Riis

Published in:
Proceedings of the 23rd International Symposium on Static Analysis (SAS 2016)

Publication date:
2016

Document Version
Peer reviewed version

A Parametric Abstract Domain for Lattice-Valued Regular Expressions

Jan Midtgaard, Flemming Nielson, and Hanne Riis Nielson

DTU Compute, Technical University of Denmark

Abstract. We present a lattice-valued generalization of regular expressions as an abstract domain for static analysis. The parametric abstract domain rests on a generalization of Brzozowski derivatives and works for both finite and infinite lattices. We develop both a co-inductive, simulation algorithm for deciding ordering between two domain elements and a widening operator for the domain. Finally we illustrate the domain with a static analysis that analyses a communicating process against a lattice-valued regular expression expressing the environment's network communication.

1 Introduction

As static analysis becomes more and more popular, so increases the need for reusable abstract domains. Within numerical abstract domains the past four decades have provided a range of such domains (signs, constant propagation, congruences, intervals, octagons, polyhedra, ...) but for non-numerical domains the spectrum is less broad (notable exceptions include abstract cofibered domains [29] and tree schemata [23]). At the same time regular languages (regular expressions and finite automata) have enabled computer scientists to create models of software systems and to reason about them both with pen-and-paper and with model checking tools.

In this paper we recast and generalize regular expressions in an abstract interpretation setting. In particular we formulate a parametric abstract domain of lattice-valued regular expressions. We illustrate the domain by extending a traditional static analysis that infers properties of the variables of a communicating process to also analyze network activity. For example, when instantiating the regular expression domain with a domain of channel names and interval values, we can express values such as $(\texttt{ask}![0;+\infty]+\texttt{report}![0;+\infty]\cdot\texttt{hsc}?[-\infty;+\infty])^*$ which describes an iterative communication pattern in which each iteration either outputs a non-negative integer on the `ask`-channel, or outputs a non-negative integer on the `report`-channel followed by reading *any value* on the `hsc`-channel. As significant amounts of modern software depend critically on message-passing network protocols and the software's ability to behave according to certain communication policies, our illustration analysis serves as a first step towards enabling static analyses to address this challenge. The resulting abstract domain grew out of this development but certainly has other applications.

$$\mathcal{L}(\emptyset) = \emptyset \qquad \mathcal{L}(r^*) = \cup_{i \geq 0}\ \mathcal{L}(r)^i \qquad \mathcal{L}(r_1 + r_2) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$$
$$\mathcal{L}(\epsilon) = \{\epsilon\} \qquad \mathcal{L}(r_1 \cdot r_2) = \mathcal{L}(r_1) \cdot \mathcal{L}(r_2) \qquad \mathcal{L}(r_1 \,\&\, r_2) = \mathcal{L}(r_1) \cap \mathcal{L}(r_2)$$
$$\mathcal{L}(\ell) = \{c \mid c \in \gamma(\ell)\} \qquad \mathcal{L}(\complement\, r) = \wp(C^*) \setminus \mathcal{L}(r)$$

Fig. 1: Denotation of lattice-valued regular expressions: $\mathcal{L} : \widehat{R}_A \longrightarrow \wp(C^*)$

The contributions of this article are as follows:

- We develop a parametric regular expression domain over finite lattices including a co-inductive ordering algorithm and a widening operator (Sec. 2),
- we generalize the constructions to infinite lattices (Sec. 3),
- we illustrate the domain with a static analysis for analyzing a communicating process (Sec. 4), and
- we report on our prototype implementation (Sec. 5).

2 Regular expressions over complete lattices

We first consider how to view lattice-valued regular expressions as a parametric abstract domain parameterized by an abstract domain A for its character literals. Let $\langle A; \sqsubseteq \rangle$ be a partially ordered set with a corresponding Galois insertion $\langle \wp(C), \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle A, \sqsubseteq \rangle$ (i.e., a Galois connection in which $\alpha : \wp(C) \longrightarrow A$ is surjective [9]) connecting A to its concrete meaning (some set of characters C). We let ℓ range over the elements of A. An element a of a lattice A is an *atom* if $\bot \sqsubset a$ and there does not exist an ℓ such that $\bot \sqsubset \ell \sqsubset a$. We write $\mathit{Atoms}(A)$ for the set of A's atom elements and let a, b, c range over these. We furthermore require $\alpha : \mathit{Atoms}(\wp(C)) \longrightarrow \mathit{Atoms}(A)$, i.e., that α maps the atoms of $\wp(C)$ (the singleton sets) to atoms of A. These assumptions have a number of consequences:

- $\langle A; \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ is a complete lattice [9, Prop. 9],
- γ is strict ($\gamma(\bot) = \emptyset$),
- $\langle A; \sqsubseteq \rangle$ is an *atomic lattice* [10] (any non-bottom element has an atom less or equal to it),
- $\langle A; \sqsubseteq \rangle$ is an *atomistic* [15] (or atomically generated) lattice: it is atomic and any non-bottom element can be written as a join of atoms.
- $\alpha : \mathit{Atoms}(\wp(C)) \longrightarrow \mathit{Atoms}(A)$ is surjective,
- Atoms have no overlapping meaning: $a \neq a' \implies \gamma(a) \cap \gamma(a') = \emptyset$

Overall the Galois insertion assumption lets A inherit the complete lattice structure of $\wp(C)$. The further assumption of atom preservation lets A further inherit the atomic and atomistic structure of $\wp(C)$. These assumptions still permit a range of known base lattices, such as signs, parity, power sets, intervals, etc. For the rest of this section we will further assume that A is finite and later in Sec. 3 lift that restriction.

The elements of A will play the role of the regular expression alphabet. Now the language of lattice-valued regular expressions is defined as follows:

$$\widehat{R}_A ::= \emptyset \mid \epsilon \mid \ell \mid \widehat{R}_A^* \mid \widehat{R}_A \cdot \widehat{R}_A \mid \complement\, \widehat{R}_A \mid \widehat{R}_A + \widehat{R}_A \mid \widehat{R}_A \,\&\, \widehat{R}_A \qquad \text{where } \ell \in A \setminus \{\bot\}$$

$$\widehat{\mathcal{D}}_a(\emptyset) = \emptyset \qquad \widehat{\mathcal{D}}_a(r_1 \cdot r_2) = \begin{cases} \widehat{\mathcal{D}}_a(r_1) \cdot r_2 + \widehat{\mathcal{D}}_a(r_2) & \epsilon \sqsubseteq_\approx r_1 \\ \widehat{\mathcal{D}}_a(r_1) \cdot r_2 & \epsilon \not\sqsubseteq_\approx r_1 \end{cases}$$

$$\widehat{\mathcal{D}}_a(\epsilon) = \emptyset \qquad \widehat{\mathcal{D}}_a(\complement\, r) = \complement\, \widehat{\mathcal{D}}_a(r)$$

$$\widehat{\mathcal{D}}_a(\ell) = \begin{cases} \epsilon & a \sqsubseteq \ell \\ \emptyset & a \not\sqsubseteq \ell \end{cases} \qquad \widehat{\mathcal{D}}_a(r_1 + r_2) = \widehat{\mathcal{D}}_a(r_1) + \widehat{\mathcal{D}}_a(r_2)$$

$$\widehat{\mathcal{D}}_a(r^*) = \widehat{\mathcal{D}}_a(r) \cdot r^* \qquad \widehat{\mathcal{D}}_a(r_1 \,\&\, r_2) = \widehat{\mathcal{D}}_a(r_1) \,\&\, \widehat{\mathcal{D}}_a(r_2)$$

Fig. 2: Lattice-valued Brzozowski derivatives: $\widehat{\mathcal{D}} : \mathit{Atoms}(A) \longrightarrow \widehat{R}_A \longrightarrow \widehat{R}_A$

Notice how we include both complement $\complement$ and intersection & in the regular expressions [6] (the result is also referred to as *extended* or *generalized regular expressions*). Fig. 1 lists our generalized denotation for the lattice-valued regular expressions. One significant difference from the traditional definition, is how we concretize lattice literals into one-element strings using the concretization function γ (traditionally, $\mathcal{L}(c) = \{c\}$ for a character c). We immediately get the traditional definition if we instantiate with $A = \wp(C)$ and the identity Galois insertion (and allow only atoms $\{c\}$ as literals). Both traditional regular expressions and lattice-valued regular expressions operate over a finite alphabet. However in contrast to traditional regular expressions where the finite alphabet carries through in the denoted language, γ may concretize a single 'lattice character' to an infinite set, e.g., $\mathcal{L}(\mathit{even}) = \{\ldots, -2, 0, 2, \ldots\}$ in a parity lattice. From the denotation it is also apparent how excluding $\bot \in A$ from lattice literals loses no generality, as bottom is expressible in the regular expressions as $\emptyset$ because γ is strict. Note that it is possible to express the same language in syntactically different ways: for example, $\emptyset$, $\emptyset \cdot \mathit{even}$, and $\mathit{even} \,\&\, \mathit{odd}$ all denote the same empty language. We therefore write $\approx$ to denote language equality in the regular expression domain.

The lattice-valued regular expressions are ordered under language inclusion: $r \sqsubseteq_\approx r' \iff \mathcal{L}(r) \subseteq \mathcal{L}(r')$. Note how we use a different symbol $\sqsubseteq_\approx$ to help distinguish the ordering of the lattice-valued regular expressions from $\sqsubseteq$, the ordering of the input domain A. The language inclusion ordering motivates our requirement for a Galois insertion: $\forall a, a' \in \mathit{Atoms}(A).\ a \sqsubseteq_\approx a' \iff \mathcal{L}(a) \subseteq \mathcal{L}(a') \iff \gamma(a) \subseteq \gamma(a') \iff a = (\alpha \circ \gamma)(a) \sqsubseteq a'$, i.e., the two orderings are compatible. The ordering is not anti-symmetric: $\emptyset \sqsubseteq_\approx \mathit{even} \,\&\, \mathit{odd}$ and $\mathit{even} \,\&\, \mathit{odd} \sqsubseteq_\approx \emptyset$ but $\emptyset \neq \mathit{even} \,\&\, \mathit{odd}$. To regain a partial order we consider elements up to language equality. The resulting regular expression domain constitutes a lattice: the least and greatest elements (bottom and top) are $\emptyset$ and $\top^*$, respectively (with $\top$ being the top element from A). Furthermore, the least upper bound and the greatest lower bound of two elements r_1 and r_2 are given symbolically by $r_1 + r_2$ and $r_1 \,\&\, r_2$, respectively. However the regular expression domain does not constitute a *complete* lattice. For example, the least upper bound of the chain $\epsilon \sqsubset \epsilon + \mathit{even} \cdot \mathit{odd} \sqsubset \epsilon + \mathit{even} \cdot \mathit{odd} + \mathit{even} \cdot \mathit{even} \cdot \mathit{odd} \cdot \mathit{odd} \sqsubset \ldots$ is an infinite sum $\sum_n \mathit{even}^n \mathit{odd}^n$ which is not a regular language over A, but context-free and there is no least regular language containing it.

$$\begin{aligned}
\mathit{nullable}(\emptyset) &= \mathit{false} & \mathit{nullable}(r_1 \cdot r_2) &= \mathit{nullable}(r_1) \wedge \mathit{nullable}(r_2) \\
\mathit{nullable}(\epsilon) &= \mathit{true} & \mathit{nullable}(\complement\, r) &= \neg \mathit{nullable}(r) \\
\mathit{nullable}(\ell) &= \mathit{false} & \mathit{nullable}(r_1 + r_2) &= \mathit{nullable}(r_1) \vee \mathit{nullable}(r_2) \\
\mathit{nullable}(r_1^*) &= \mathit{true} & \mathit{nullable}(r_1 \,\&\, r_2) &= \mathit{nullable}(r_1) \wedge \mathit{nullable}(r_2)
\end{aligned}$$

Fig. 3: The *nullable* operation: $\mathit{nullable} : \widehat{R}_A \longrightarrow \mathit{Bool}$

As a fundamental operation over the lattice-valued regular expressions, we consider the Brzozowski derivative [6]. A traditional Brzozowski derivative of a regular expression r with respect to some character c returns a regular expression denoting the suffix strings w resulting from having read a character c from r: $\mathcal{L}(\mathcal{D}_c(r)) = c\backslash\mathcal{L}(r) = \{w \mid c \cdot w \in \mathcal{L}(r)\}$. In Fig. 2 we define the generalized lattice-based derivatives. Note how we derive lattice-valued regular expressions only with respect to lattice atoms. Brzozowski's derivatives gave rise to a central equation, which also holds for the lattice-valued generalization: all regular expressions can be expressed as a sum of derivatives (modulo an optional epsilon):

Theorem 1 (Sum of derivatives [6]).

$$\forall r \in \widehat{R}_A.\ \ r \ \approx \sum_{a \in \mathit{Atoms}(A)} a \cdot \widehat{\mathcal{D}}_a(r) \ + \ \delta(r) \quad \textit{where } \delta(r) = \begin{cases} \epsilon & \epsilon \sqsubseteq_\approx r \\ \emptyset & \epsilon \not\sqsubseteq_\approx r \end{cases}$$

The proof utilizes that atoms are non-overlapping. We can use the equation to characterize the lattice-valued Brzozowski derivatives. We first generalize the notation to account for sets in the denotations: $cs\backslash L = \{w \mid \forall c \in cs.\ c \cdot w \in L\}$ and then utilize this notation in the characterization.

Lemma 2 (Meaning of derivatives).

$$\forall r \in \widehat{R}_A, a \in \mathit{Atoms}(A).\ \ \mathcal{L}(\widehat{\mathcal{D}}_a(r)) = \gamma(a)\backslash\mathcal{L}(r) = \{w \mid \forall c \in \gamma(a).\ c \cdot w \in \mathcal{L}(r)\}$$

Based on the inclusion ordering and Lemma 2 one can easily verify that $\widehat{\mathcal{D}}$ is monotone in the second, regular expression parameter.

Lemma 3 ($\widehat{\mathcal{D}}$ monotone). $\forall a \in \mathit{Atoms}(A), r, r' \in \widehat{R}_A.\ r \sqsubseteq_\approx r' \implies \widehat{\mathcal{D}}_a(r) \sqsubseteq_\approx \widehat{\mathcal{D}}_a(r')$

We test the side-condition $\epsilon \sqsubseteq_\approx r_1$ of Fig. 2 with a dedicated procedure, *nullable*, defined in Fig. 3. We can prove that *nullable* has the intended meaning:

Lemma 4 (*nullable* correct). $\forall r \in \widehat{R}_A.\ \ \epsilon \sqsubseteq_\approx r \iff \mathit{nullable}(r)$

We can extend the definition of derivatives to sequences of derivatives. Sequences are defined inductively: $s ::= \epsilon \mid as$ and a derivation with respect to a sequence is defined structurally [6]: $\widehat{\mathcal{D}}_\epsilon(r) = r$ and $\widehat{\mathcal{D}}_{as}(r) = \widehat{\mathcal{D}}_s(\widehat{\mathcal{D}}_a(r))$ meaning that $\widehat{\mathcal{D}}_{a_1 \ldots a_n}(r) = \widehat{\mathcal{D}}_{a_n}(\ldots \widehat{\mathcal{D}}_{a_1}(r))$.

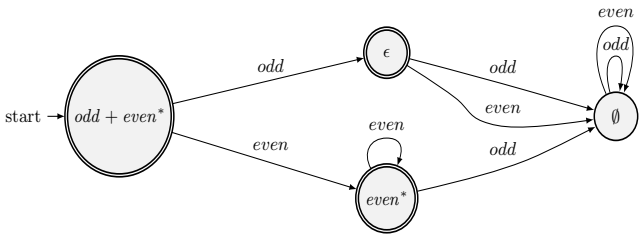

Fig. 4: Example automaton derived for $odd + even^*$

2.1 Derivatives as automata

One can view Brzozowski derivatives as a means for translating a regular expression to an automaton [6]. This view extends to the lattice-valued generalization: (1) Each state is identified with a regular expression denoting the language it accepts, (2) There is a transition consuming a from one state, r, to the state $\widehat{\mathcal{D}}_a(r)$, and (3) A state r is accepting if and only if $nullable(r)$. Consider the regular expression $odd + even^*$. Since $\widehat{\mathcal{D}}_{even}(odd + even^*) \approx even^*$ the corresponding automaton (depicted in Fig.4) can transition from the former to the latter by consuming the atom $even$. The state corresponding to the root expression $odd + even^*$ furthermore acts as the initial state. As $odd + even^*$ is also $nullable$ the corresponding state is also a final state.

Brzozowski [6] proved that there are only a bounded number of different derivatives of a given regular expression up to associativity, commutivity, and idempotence (ACI) of $+$.[1] Intuitively, ACI of $+$ means that the terms of a sum act as a set: parentheses and term order are irrelevant and any term present is syntactically unique. For the lattice-valued generalization we can similarly establish an upper bound by structural induction on r as a counting argument on the number of syntactically unique term elements:

Lemma 5 (Number of dissimilar derivatives).

$$\forall r \in \widehat{R}_A.\ \exists n.\ |\{\widehat{\mathcal{D}}_s(r) \mid s \in Atoms(A)^*\}_{=_{ACI}}| \leq n$$

As a consequence a resulting automaton is guaranteed to have only a finite number of states. However a resulting automaton is not necessarily minimal. By incorporating additional *simplifying reductions* ($\epsilon \cdot r = r = r \cdot \epsilon$, $\emptyset + r = r$, ...) we can identify more equivalent states in a resulting automaton, thereby reducing its size further and thus making the approach practically feasible [25]. For example, there are five dissimilar derivatives (including the root expression) of $odd + even^*$ up to ACI of $+$:

$$\widehat{\mathcal{D}}_{even}(odd + even^*) = \emptyset + \epsilon \cdot even^* \qquad \widehat{\mathcal{D}}_{even}(\emptyset + \epsilon \cdot even^*) = \emptyset + \emptyset \cdot even^* + \epsilon \cdot even^*$$
$$\widehat{\mathcal{D}}_{odd}(odd + even^*) = \epsilon + \emptyset \cdot even^* \qquad \widehat{\mathcal{D}}_{even}(\epsilon + \emptyset \cdot even^*) = \emptyset + \emptyset \cdot even^*$$

[1] It has later been pointed out [27, 26, 14] that Brzozowski's proof had a minor flaw, that could be fixed by patching the statement of the theorem [27] or by patching the definition of derivatives to avoid the syntactic occurrence of δ [26]. We have followed the latter approach in our generalization.

```
1  proc leq_test(r1, r2)
2      memo_tbl := {}
3      proc leq_memo(r1, r2)
4          if (r1, r2) ∈ACI memo_tbl then return
5          memo_tbl := memo_tbl ∪ {(r1, r2)}
6          if nullable(r1) ⟹ nullable(r2)
7          then for all a ∈ Atoms(A), leq_memo(D̂a(r1), D̂a(r2))
8                   return
9          else raise False
10     try leq_memo(r1, r2) with False => return false
11     return true
```

Fig. 5: Ordering algorithm

Any remaining derivatives, e.g., $\widehat{\mathcal{D}}_{odd}(\emptyset + \epsilon \cdot even^*) = \emptyset + \emptyset \cdot even^* + \emptyset \cdot even^*$ is ACI equivalent to one of these: $\emptyset + \emptyset \cdot even^* + \emptyset \cdot even^* =_{ACI} \emptyset + \emptyset \cdot even^*$. By additional simplifying reductions, the five derivatives can be reduced to four: $odd + even^*$, $even^*$, ϵ, $\emptyset$ respectively. Collectively, these four derivatives represent the four states of Fig. 4 (including the explicit error state $\emptyset$). We stress that the results of this paper require only the ACI equivalences. For readability and to make the techniques practical, for the rest of the article we will incorporate the further simplifying reductions, which we denote $ACI+$.

2.2 An ordering algorithm

Both the least upper bound and the greatest lower bound of two regular expressions r_1 and r_2 can be symbolically represented as $r_1 + r_2$ and $r_1 \,\&\, r_2$, respectively. However a procedure for deciding domain ordering is not as easy: The language inclusion ordering $\sqsubseteq$ is ideal for pen-and-paper results, but it is not a tractable approach for algorithmically comparing elements on a computer. We will therefore develop an algorithm based on derivatives.

With the "derivatives-as-automata-states" in mind, we formulate in Fig. 5 a procedure for computing a (constructive) simulation. Essentially, the algorithm corresponds to lazily exploring each state of the two regular expressions' automata using Brzozowski's construction, and computing a simulation (implemented as a hashtable) between these state pairs. Upon successful termination the algorithm will have computed in `memo_tbl` a simulation between the derivatives of r_1 and r_2. For example, if we invoke the algorithm with arguments $(\epsilon, even^*)$ it will compute a simulation $\texttt{memo\_tbl} = \{(\epsilon, even^*), (\emptyset, \emptyset), (\emptyset, even^*)\}$ and ultimately return true. Underway, the first call to `leq_memo` with, e.g., arguments $(\emptyset, \emptyset)$ will memorize the pair and after ensuring that $false \implies false$ it will recursively call `leq_memo` with both $(\widehat{\mathcal{D}}_{odd}(\emptyset), \widehat{\mathcal{D}}_{odd}(\emptyset)) = (\emptyset, \emptyset)$ and $(\widehat{\mathcal{D}}_{even}(\emptyset), \widehat{\mathcal{D}}_{even}(\emptyset)) = (\emptyset, \emptyset)$ as arguments. These two invocations will immediately return successfully due to the memorization. By memorizing each pair of regular expressions and testing `memo_tbl` for membership up to ACI equivalence the algorithm is guaranteed to terminate. As such, the algorithm is an

inclusion (or *containment*) analogue of Grabmayer's co-inductive axiomatization of regular expression equivalence [14, 17]. There are different opportunities for optimizing the ordering algorithm. By reflexivity, we can avoid derivatives when `leq_memo` is invoked with equal arguments r_1 and r_2. Another possibility is to utilize *hash consing* to avoid computing the same derivatives repeatedly.

We can show that the language inclusion ordering and the derivative-based ordering are in fact equivalent. The proof utilizes both Theorem 1, $\widehat{\mathcal{D}}$'s monotonicity (Lemma 3), and the correctness of *nullable* (Lemma 4). To prove equivalence we consider a simulation ordering which we will use as a stepping stone. We define a simulation $\preceq$ to be a relation that satisfies $r \preceq r'$ iff $\mathit{nullable}(r) \implies \mathit{nullable}(r')$ and for all atoms a: $\widehat{\mathcal{D}}_a(r) \preceq \widehat{\mathcal{D}}_a(r')$. We can view such a simulation $\preceq$ as a fixed point of a function $F : \wp(\widehat{R}_A \times \widehat{R}_A) \longrightarrow \wp(\widehat{R}_A \times \widehat{R}_A)$ defined as follows:

$$F(\preceq') = \{(r_1, r_2) \mid \mathit{nullable}(r_1) \implies \mathit{nullable}(r_2)\} \\ \cap \{(r_1, r_2) \mid \forall \text{ atoms } a : (\widehat{\mathcal{D}}_a(r_1), \widehat{\mathcal{D}}_a(r_2)) \in \preceq'\}$$

It is now straight-forward to verify that F is monotone. Since it is defined over a complete lattice (sets of regular expression pairs), the greatest fixed point is well-defined by Tarski's fixed-point theorem. In particular, for a fixed point $F(\preceq) = \preceq$, we then have $\preceq = \{(r_1, r_2) \mid \mathit{nullable}(r_1) \implies \mathit{nullable}(r_2)\} \cap \{(r_1, r_2) \mid \forall \text{ atoms } a : (\widehat{\mathcal{D}}_a(r_1), \widehat{\mathcal{D}}_a(r_2)) \in \preceq\}$. Write $\dot{\preceq}$ for $\operatorname{gfp} F$. We are now in position to prove equivalence of the language inclusion ordering and the derivative-based ordering. In the following lemma we do so in two steps:

Lemma 6 (Ordering equivalence).

(a) $\forall r, r' \in \widehat{R}_A.$ `leq_test`(r, r') *returns true* $\iff r \dot{\preceq} r'$
(b) $\forall r, r' \in \widehat{R}_A.\ r \sqsubseteq r' \iff r \dot{\preceq} r'$

Algorithms for testing inclusion (or containment) of two regular expressions r_1 and r_2 are well known [22]. The textbook algorithm tests $r_1 \,\&\, \complement\, r_2$ for emptiness [22]. In our generalized setting, this would correspond to invoking `leq_test`$(r_1 \,\&\, \complement\, r_2, \emptyset)$, for which all sub-derivatives of the second parameter $\emptyset$ passed around by `leq_memo` continue to be $\emptyset$ (which is clearly not *nullable*). The loop would thereby explore all paths from the first parameter's root expression $r_1 \,\&\, \complement\, r_2$ to a *nullable* derivative, corresponding to a search for a reachable acceptance state in a corresponding DFA under the derivatives-as-automata-view. As such, the textbook emptiness-testing algorithm can be viewed as a special case of our derivative-based algorithm.

2.3 Widening

A static analysis based on Kleene iteration over the regular expression domain is not guaranteed to terminate, as it contains infinite, strictly increasing chains: $\epsilon \sqsubset \epsilon + l \sqsubset \epsilon + l + (l \cdot l) \sqsubset \epsilon + l + (l \cdot l) + (l \cdot l \cdot l) \sqsubset \ldots$. For this reason we need a widening operator. From a high-level point of view the widening operator works by (a) formulating an equation system, (b) collapsing some of the equations in

order to avoid infinite, strictly increasing chains, and (c) solving the collapsed equation system to get back a regular expression. Step (b) is inspired by a widening operator of Feret [11] and Le Gall, Jeannet, and Jéron [13], and step (c) uses a translation scheme due to Brzozowski [6].

Let us consider an example of widening odd and $even^*$. As a first step we form their sum: $odd + even^*$. We know from Sec. 2.1 that it has four different simplified derivatives: $\{odd + even^*, even^*, \epsilon, \emptyset\}$. By Theorem 1 we can characterize them as equations, where we name the four derivatives R_0, R_1, R_2, R_3 (R_0 denotes the root expression):

$$\begin{aligned} R_0 &\approx even \cdot R_1 + odd \cdot R_2 + \epsilon \\ R_1 &\approx even \cdot R_1 + odd \cdot R_3 + \epsilon \\ R_2 &\approx even \cdot R_3 + odd \cdot R_3 + \epsilon \\ R_3 &\approx even \cdot R_3 + odd \cdot R_3 \end{aligned}$$

We subsequently collect the coefficients of R_0, R_1, R_2, R_3 and state the resulting equation system as a matrix. For example, by collecting the coefficients to R_3 in the equation $R_2 \approx even \cdot R_3 + odd \cdot R_3 + \epsilon$ we obtain $R_2 \approx (even + odd) \cdot R_3 + \epsilon$. The resulting matrix describes the transitions of a corresponding finite automata as displayed in Fig. 4:

$$\begin{bmatrix} R_0 \\ R_1 \\ R_2 \\ R_3 \end{bmatrix} \approx \begin{bmatrix} \emptyset & even & odd & \emptyset \\ \emptyset & even & \emptyset & odd \\ \emptyset & \emptyset & \emptyset & even + odd \\ \emptyset & \emptyset & \emptyset & even + odd \end{bmatrix} \cdot \begin{bmatrix} R_0 \\ R_1 \\ R_2 \\ R_3 \end{bmatrix} + \begin{bmatrix} \epsilon \\ \epsilon \\ \epsilon \\ \emptyset \end{bmatrix}$$

The widening operator partitions the set of derivatives into a fixed, finite number of equivalence classes and works for any such partitioning. In the present case we will use a *coloring function*, $col : \widehat{R}_A \longrightarrow \widehat{R}_A \longrightarrow [1;3]$ to partition a set of derivatives with respect to a given root expression r':

$$col_{r'}(r) = \begin{cases} 1 & \text{if } r =_{ACI} r' \\ 2 & \text{if } r \neq_{ACI} r' \text{ and } nullable(r) \\ 3 & \text{if } r \neq_{ACI} r' \text{ and } \neg nullable(r) \end{cases}$$

$col_{odd+even^*}$ will thus induce a partitioning: $\{\overbrace{\{odd + even^*\}}^{\text{color 1}}, \overbrace{\{even^*,\ \epsilon\}}^{\text{color 2}}, \overbrace{\{\emptyset\}}^{\text{color 3}}\}$. This partitioning can be expressed by equating R_1 and R_2. By adding the right-hand-sides of R_1 and R_2 into a combined right-hand-side for their combination R_{12}, we can be sure that the least solution to R_{12} in the resulting equation system is also a solution to the variables R_1 and R_2 in the original equation system. For example, the equation $R_0 \approx even \cdot R_1 + odd \cdot R_2 + \epsilon$ becomes $R_0 \approx (even + odd) \cdot R_{12} + \epsilon$ in the collapsed system. The resulting equation system now reads:

$$\begin{bmatrix} R_0 \\ R_{12} \\ R_3 \end{bmatrix} \approx \begin{bmatrix} \emptyset & even + odd & \emptyset \\ \emptyset & even & even + odd \\ \emptyset & \emptyset & even + odd \end{bmatrix} \cdot \begin{bmatrix} R_0 \\ R_{12} \\ R_3 \end{bmatrix} + \begin{bmatrix} \epsilon \\ \epsilon \\ \emptyset \end{bmatrix}$$

This particular step of the algorithm represents a potential information loss, as the coefficients of each of R_1 and R_2 are merged into joint coefficients for

ALGORITHM WIDENING(r, r')

1. Form $r_0 = r + r'$
2. Derive the characteristic equations over variables R_i:
   $R_i \approx \sum_{a_j \in Atoms(A)} a_j \cdot R_j + \delta(r_i)$
3. For each equation collect the coefficients for each variable R_i
4. Compute equivalence classes for R_i
5. Collapse equations based on equivalence classes and solve the collapsed equations
6. Return the solution to (the equivalence class containing) R_0

Fig. 6: The widening algorithm

R_{12}. We can now solve these by combining (a) elimination of variables and (b) Arden's lemma [3] (which states that an equation of the form $X \approx A \cdot X + B$ has solution $X \approx A^* \cdot B$). The equation $R_3 \approx (even + odd) \cdot R_3 + \emptyset$ therefore has solution $R_3 \approx (even + odd)^* \cdot \emptyset =_{ACI+} \emptyset$, and we can thus eliminate the variable R_3 by substituting this solution in (and simplifying):

$$\begin{bmatrix} R_0 \\ R_{12} \end{bmatrix} \approx \begin{bmatrix} \emptyset & even + odd \\ \emptyset & even \end{bmatrix} \cdot \begin{bmatrix} R_0 \\ R_{12} \end{bmatrix} + \begin{bmatrix} \epsilon \\ \epsilon \end{bmatrix}$$

Now $R_{12} \approx even \cdot R_{12} + \epsilon$ has solution $R_{12} \approx even^* \cdot \epsilon =_{ACI+} even^*$ by Arden's lemma. Again we eliminate the variable:

$$\left[R_0\right] \approx \left[\emptyset\right] \cdot \left[R_0\right] + \left[(even + odd) \cdot even^* + \epsilon\right]$$

From this we read off the result: $R_0 \approx (even + odd) \cdot even^* + \epsilon$. which clearly includes both arguments odd and $even^*$ to the widening operator as well as some additional elements, such as $odd \cdot even$.

We summarize the widening algorithm in Fig. 6 where we write R_i for the variable corresponding to the derivative regular expression r_i. We can furthermore prove that the procedure indeed is a widening operator.

Theorem 7. *The widening algorithm constitutes a widening operator:*

(a) *the result is greater or equal to any of the arguments and*
(b) *given an increasing chain $r_0 \sqsubseteq_{\approx} r_1 \sqsubseteq_{\approx} r_2 \sqsubseteq_{\approx} \ldots$ the resulting widening sequence defined as $\overline{r}_0 = r_0$ and $\overline{r}_{k+1} = \overline{r}_k \triangledown r_{k+1}$ stabilizes after a finite number of steps.*

The widening algorithm in Fig. 6 works for any partitioning into a fixed number of equivalence classes. The above example illustrates the setting (level 0) in which a coloring function is used directly to partition the derivatives into three equivalence classes. Inspired by Feret [11] and Le Gall, Jeannet, and Jéron [13], we generalize this pattern to distinguish two regular expressions at level $k+1$ if their derivatives can be distinguished at level k:

$$r_1 \approx_0^{col_r} r_2 \text{ iff } col_r(r_1) = col_r(r_2)$$
$$r_1 \approx_{k+1}^{col_r} r_2 \text{ iff } r_1 \approx_k^{col_r} r_2 \wedge \forall \text{ atoms } a.\ \widehat{\mathcal{D}}_a(r_1) \approx_k^{col_r} \widehat{\mathcal{D}}_a(r_2)$$

$$
\begin{aligned}
\widehat{range}(\emptyset) &= \widehat{to\_equivs}(\top) \\
\widehat{range}(\epsilon) &= \widehat{to\_equivs}(\top) \\
\widehat{range}(\ell) &= \widehat{to\_equivs}(\ell) \\
\widehat{range}(r^*) &= \widehat{range}(r)
\end{aligned}
\qquad
\begin{aligned}
\widehat{range}(r_1 \cdot r_2) &= \begin{cases} \widehat{overlay}(\widehat{range}(r_1), \widehat{range}(r_2)) & \epsilon \sqsubseteq r_1 \\ \widehat{range}(r_1) & \epsilon \not\sqsubseteq r_1 \end{cases} \\
\widehat{range}(\complement\, r) &= \widehat{range}(r) \\
\widehat{range}(r_1 + r_2) &= \widehat{overlay}(\widehat{range}(r_1), \widehat{range}(r_2)) \\
\widehat{range}(r_1 \,\&\, r_2) &= \widehat{overlay}(\widehat{range}(r_1), \widehat{range}(r_2))
\end{aligned}
$$

Fig. 7: Generic $\widehat{range}$ function for partitioning A's atoms: $\widehat{range} : \widehat{R}_A \longrightarrow \widehat{equiv}_A$

The resulting partitioning $\approx_k^{col_r}$ essentially expresses bisimilarity up to some bound k. With this characterization in mind, we define an extensive, idempotent operator $\rho_k^{col_r}$ that quotients the language of the underlying languages with respect to $\approx_k^{col_r}$: $r \triangledown r' = \rho_k^{col_{r+r'}}(r+r')$. Collectively, $\rho_k^{col_{r+r'}}$ represents a family of widening operators (one for each choice of k).

In our example of widening *odd* and $even^*$ the coloring function assigns the error state R_3 (representing $\emptyset$) to a different equivalence class than any non-error states, thereby preventing them from being collapsed. Such collapsing will result in a severe precision loss, as the self-loops of error states such as R_3 are inherited by a resulting collapsed state, thereby leading to spurious self-loops in the result. After having identified the issue on a number of examples, we designed a refined coloring function $col_{alt} : \widehat{R}_A \longrightarrow \widehat{R}_A \longrightarrow [1;4]$ that gives a separate color 4 to "error states": regular expressions from which a *nullable* expression is unreachable under any sequence of derivatives. In the matrix representation such expressions can be identified by their complement: we can find all non-error states by a depth-first marking of all R_i (representing r_i) reachable from a *nullable* state under a reverse ordering of the derivative transitions.[2]

3 From finite to infinite lattices

The Brzozowski identity and the algorithms utilizing it are only tractable up to a certain point: For example, even for a finite interval lattice over 32-bit integers, there are 2^{32} atoms of the shape $[i;i]$ making a sum (and loops iterating) over all such atoms unrealistic to work with. In fact, many derivatives are syntactically identical, which allow us to consider only a subset of "representative" atoms. For example, consider a derivative over interval-valued regular expressions: $\widehat{\mathcal{D}}_{[1;1]}([1;10] + [20;22]) = \epsilon + \emptyset$. Clearly the result is identical for atoms $[2;2], \ldots, [10;10]$.[3] To this end we seek to partition a potentially infinite set of

[2] Solving the equations for such error states before step 5 (collapsing) has the same effect: their collective solution is $\emptyset$ in the matrix, and substituting the solution in removes any transitions to and from them, and thereby any observable effect of grouping an error state and a non-error state in the same equivalence class.

[3] The result is also $\epsilon + \emptyset$ for $[20;20], [21;21], [22;22]$ up to ACI of $+$, but that just constitutes a refinement identifying even more equivalent atoms.

atoms into a finite set of equivalence classes $[a_1], \ldots, [a_n]$ with identical derivatives. We represent a partition as an abstract type $\widehat{equiv}_A$ suitably instantiated for each lattice A. One operation $\widehat{to\_equivs} : A \longrightarrow \widehat{equiv}_A$ computes a partition for a given lattice literal and a second operation $\widehat{overlay} : \widehat{equiv}_A \longrightarrow \widehat{equiv}_A \longrightarrow \widehat{equiv}_A$ combines two partitions into a refined one. The computed partition should satisfy the following two properties:

$$\forall \ell \in A, [a_i] \in \widehat{to\_equivs}(\ell), a, a' \in Atoms(A).$$
$$a, a' \in [a_i] \implies (a \sqsubseteq \ell \wedge a' \sqsubseteq \ell) \vee (a \not\sqsubseteq \ell \wedge a' \not\sqsubseteq \ell) \quad (1)$$
$$\forall \overline{[b]}, \overline{[c]} \in \widehat{equiv}_A, [a_i] \in \widehat{overlay}(\overline{[b]}, \overline{[c]}), a, a' \in Atoms(A).$$
$$a, a' \in [a_i] \implies \exists j, k.\ a, a' \in [b_j] \wedge a, a' \in [c_k] \quad (2)$$

where $\overline{[b]}, \overline{[c]}$ range over partitions of A's atoms. Based on $\widehat{to\_equivs}$ and $\widehat{overlay}$ we can formulate in Fig. 7 a generic $\widehat{range}$ function, that computes an atom partition for a given regular expression. Assuming that $\widehat{to\_equivs}$ produces a partition and that $\widehat{overlay}$ preserves partitions the result of $\widehat{range}$ will also be a partition. Specifically, $\widehat{range}$ computes a partition over A's atoms for the equivalence relation $\widehat{\mathcal{D}}_a(r) = \widehat{\mathcal{D}}_{a'}(r)$. We can verify this property by structural induction over r:

Lemma 8. $\forall r \in \widehat{R}_A, [a_i] \in \widehat{range}(r), a, a' \in Atoms(A).\, a, a' \in [a_i] \implies \widehat{\mathcal{D}}_a(r) = \widehat{\mathcal{D}}_{a'}(r)$

As a consequence we can optimize the ordering algorithm in Fig. 5. For all atoms a, a' such that $[a_1], \ldots, [a_n] = \widehat{overlay}(\widehat{range}(r_1), \widehat{range}(r_2))$ and $a, a' \in [a_i]$ for some $1 \leq i \leq n$, by Lemma 8 and Property 2 we have both $\widehat{\mathcal{D}}_a(r_1) = \widehat{\mathcal{D}}_{a'}(r_1)$ and $\widehat{\mathcal{D}}_a(r_2) = \widehat{\mathcal{D}}_{a'}(r_2)$ and can therefore just check one representative from each equivalence class. We thus replace line number 7 in Fig. 5 with:

then for all $[a_i] \in \widehat{overlay}(\widehat{range}(r_1), \widehat{range}(r_2))$, `leq_memo`$(\widehat{\mathcal{D}}_{\widehat{repr}([a_i])}(r_1), \widehat{\mathcal{D}}_{\widehat{repr}([a_i])}(r_2))$

where the function $\widehat{repr}$ returns a representative atom a_i from the equivalence class $[a_i]$.

Corollary 9 (Correctness of modified ordering algorithm).

$$\forall r, r' \in \widehat{R}_A.\ \texttt{leq\_test}'(r, r') \text{ returns true} \iff r \sqsubseteq_{\lesssim} r'$$

Similarly we can adjust step 2 of the widening algorithm in Fig. 6 to form finite characteristic equations. We do so by limiting the constructed sums to one term per equivalence class in $\widehat{range}$'s partition of $Atoms(A)$: $R_i \approx \sum_{[a_j] \in \widehat{range}(r_i)} \widehat{project}([a_j]) \cdot R_j + \delta(r_i)$ where $\widehat{project}([a_j])$ returns a lattice value from A accounting for all atoms in the equivalence class $[a_j]$: $\forall a \in [a_j].\ a \sqsubseteq \widehat{project}([a_j])$.[4] For infinite lattices A not satisfying ACC, we cannot ensure stabilization over, e.g., $\emptyset \sqsubset [0;0] \sqsubset [0;1] \sqsubset \ldots$ (injected as character literals into $\widehat{R}_{Interval}$), as the widening algorithm does not incorporate widening over A. However, when limited to chains with only a finite number of different lattice literals the operator constitutes a widening:

[4] Generally the solution to this equation is an over-approximation but so is the result of widening.

$$\widehat{to\_equivs}([l;u]) = \begin{cases} [-\infty;+\infty] & l=-\infty \wedge u=+\infty \\ [-\infty;u],[u+1;+\infty] & l=-\infty \wedge u\neq+\infty \\ [-\infty;l-1],[l;+\infty] & l\neq-\infty \wedge u=+\infty \\ [-\infty;l-1],[l;u],[u+1;+\infty] & l\neq-\infty \wedge u\neq+\infty \end{cases}$$

$$\widehat{overlay}([l_1;+\infty],[l_2;+\infty]) = [l_1;+\infty] \qquad l_1 = l_2 \text{ holds as an invariant}$$

$$\widehat{overlay}([l_1;u_1]::R_1',[l_2;u_2]::R_2') = \begin{cases} [l_1;u_1]::\widehat{overlay}(R_1',R_2') & l_1=l_2 \wedge u_1=u_2 \\ [l_1;u_1]::\widehat{overlay}(R_1',[u_1+1;u_2]::R_2') & l_1=l_2 \wedge u_1<u_2 \\ [l_2;u_2]::\widehat{overlay}([u_2+1;u_1]::R_1',R_2') & l_1=l_2 \wedge u_1>u_2 \end{cases}$$

Fig. 8: $\widehat{to\_equivs}$ and $\widehat{overlay}$ for the interval lattice

Corollary 10. *The modified widening algorithm constitutes a widening operator over increasing chains containing only finitely many lattice literals from A.*

The widening operator over the lattice-valued regular expressions does not incorporate a widening operator over A. As such there may be infinite, strictly increasing chains of values from A that flow into $\widehat{R}_A$ (when such values are injected as character literals). Furthermore there may be a complex flow of values from A and into $\widehat{R}_A$ and back again from $\widehat{R}_A$ and into A via $\widehat{project}$. Following abstract interpretation tradition [4], any such cyclic flows of values (be it over A or $\widehat{R}_A$) should cross at least one widening operator, e.g., on loop headers, to guarantee termination. An analysis component over A (e.g., for interval analysis of variables) that supplies $\widehat{R}_A$ with injected values from A will therefore itself have to incorporate widening over A at these points. In this situation, thanks to A's widening operator only a finite number of different values from A can flow to (the chains of) the regular expressions. We thereby satisfy Corollary 10's condition and ensure overall termination by "delegating the termination responsibility" to each of the participating abstract domains. Next, we turn to specific instances for A.

3.1 Small, finite instantiations

Simple finite lattices such as the parity lattice can meet the above interface by letting each atom a represent a singleton equivalence class $[a]$. We can then represent $\widehat{equiv}_A$ as a constant list of such atoms. For example, for the parity lattice we can implement $\widehat{to\_equivs}$ and $\widehat{overlay}$ as constant functions, returning $[even],[odd]$. It follows that $\widehat{to\_equivs}$ produces a partition, that $\widehat{overlay}$ preserves it, and that the definitions satisfies Properties 1,2.

3.2 The interval lattice

For an interval lattice [7] we can represent each equivalence class as an interval and the entire partition as a finite set of non-overlapping intervals: $\widehat{equiv}_{Interval} =$

$\wp(\mathit{Interval})$. We can formulate $\widehat{\mathit{to\_equivs}}$ in Fig. 8 as a case dispatch that takes into account the limit cases $-\infty$ and $+\infty$ of an interval literal $[l; u]$. As an example, $\widehat{\mathit{to\_equivs}}([0; 2])$ returns the partition $[-\infty; -1], [0; 2], [3; +\infty]$ of the atoms $[i; i]$. By sorting the equivalence classes (intervals) we can $\widehat{\mathit{overlay}}$ two partitions in linear time. In Fig. 8 we formulate $\widehat{\mathit{overlay}}$ as a recursive function over two such sorted partitions. The implementation satisfies the invariant that at a recursive invocation (a) neither of its arguments are empty, (b) the two leftmost lower bounds are identical, and (c) the two rightmost upper bounds are $+\infty$. As such each recursive invocation of $\widehat{\mathit{overlay}}$ combines two partitions from (their common) leftmost lower bound to $+\infty$. As an example, $\widehat{\mathit{overlay}}$ of $[-\infty; -1], [0; 2], [3; +\infty]$ and $[-\infty; -3], [-2; 1], [2; +\infty]$ returns the partition $[-\infty; -3], [-2; -1], [0; 1], [2; 2], [3; +\infty]$. We prove that the definitions have the desired properties. For $\widehat{\mathit{overlay}}$ they follow by well-ordered induction under the termination measure "number of overlapping interval pairs".

Lemma 11. *(a) $\widehat{\mathit{to\_equivs}}$ computes a partition and (b) $\widehat{\mathit{overlay}}$ preserves partitions*

Lemma 12. *$\widehat{\mathit{to\_equivs}}$ and $\widehat{\mathit{overlay}}$ satisfy Properties 1, 2*

3.3 Product lattices

We can combine partitions to form partitions over product lattices of either of the two traditional forms: Cartesian products and reduced/smash products.

The Cartesian product lattice Given two potentially infinite lattices A, B and their product lattice $A \times B$ ordered componentwise, we can partition their atoms in a compositional manner. Assuming the product lattice $A \times B$ satisfies the requirements of Sec. 2, this implicitly means we work over a domain where for all $\ell_A \in A \setminus \{\bot\}$. $\gamma(\langle \ell_A, \bot\rangle) \neq \emptyset$ and for all $\ell_B \in B \setminus \{\bot\}$. $\gamma(\langle \bot, \ell_B\rangle) \neq \emptyset$ since either would mean that, e.g., $\gamma(\langle \ell_A, \bot\rangle) = \emptyset = \gamma(\bot)$ and thereby break the Galois insertion requirement. With this implicit assumption in place, the atoms of the product lattice must be of the shape $\langle a, \bot\rangle \in \mathit{Atoms}(A) \times B$ and $\langle \bot, b\rangle \in A \times \mathit{Atoms}(B)$. Given representations $\widehat{\mathit{equiv}}_A$ and $\widehat{\mathit{equiv}}_B$ partitioning A's and B's atoms, we can partition the atoms of $A \times B$ with a partition $\widehat{\mathit{equiv}}_A \times \widehat{\mathit{equiv}}_B$ where the first component partitions atoms in $\mathit{Atoms}(A) \times \{\bot\}$ and the second component partitions atoms in $\{\bot\} \times \mathit{Atoms}(B)$. Based on operations $\widehat{\mathit{to\_equivs}}_A$ and $\widehat{\mathit{to\_equivs}}_B$ we can therefore write $\widehat{\mathit{to\_equivs}}(\ell_A, \ell_B) = (\widehat{\mathit{to\_equivs}}_A(\ell_A), \widehat{\mathit{to\_equivs}}_B(\ell_B))$. For example, for a Cartesian product of intervals $\widehat{\mathit{to\_equivs}}([-1; 2], [0; 1])$ returns the partition $(([-\infty; -2], [-1; 2], [3; +\infty]), ([-\infty; -1], [0; 1], [2; +\infty]))$. Let $\overline{[a]}$ and $\overline{[b]}$ range over partitions of A's and B's atoms. We can also write $\widehat{\mathit{overlay}}$ compositionally: $\widehat{\mathit{overlay}}((\overline{[a]}, \overline{[b]}), (\overline{[a']}, \overline{[b']})) = (\widehat{\mathit{overlay}}_A(\overline{[a]}, \overline{[a']}), \widehat{\mathit{overlay}}_B(\overline{[b]}, \overline{[b']}))$.

The reduced/smash product lattice If for two lattices A and B, for all $\ell_A \in A \setminus \{\bot\}$. $\gamma(\langle \ell_A, \bot \rangle) = \emptyset$ and for all $\ell_B \in B \setminus \{\bot\}$. $\gamma(\langle \bot, \ell_B \rangle) = \emptyset$ we can instead consider the reduced/smash product: $A * B = \{\langle \bot, \bot \rangle\} \cup (A \setminus \{\bot\}) \times (B \setminus \{\bot\})$ where atoms are of the shape $\langle a, b \rangle \in \mathit{Atoms}(A) \times \mathit{Atoms}(B)$. Again we can partition the atoms of $A*B$ with a product $\widehat{\mathit{equiv}}_A \times \widehat{\mathit{equiv}}_B$ this time interpreting an equivalence class $([a], [b]) \in \widehat{\mathit{equiv}}_A \times \widehat{\mathit{equiv}}_B$ as all atoms (a', b') where $a' \in [a]$ and $b' \in [b]$. Despite the different interpretation, we define $\widehat{\mathit{to\_equivs}}$ and $\widehat{\mathit{overlay}}$ as in Cartesian products.

Coarser partitions are possible. For the reduced/smash product we have experimented with a functional partition $[\mathit{Atoms}(A)] \longrightarrow \widehat{\mathit{equiv}}_B$ maintaining individual partitions of B for each equivalence class of $\mathit{Atoms}(A)$. For example, for a interval pair literal $\langle [1;1], [0;2] \rangle$ the coarser functional partition will have only 5 equivalence classes (both the $[-\infty; 0]$ and $[2; +\infty]$ entries map to the partition $[-\infty; +\infty]$ and the atom partition for $[0; 2]$ at entry $[1; 1]$ has three entries) whereas the finer partition will have $3 \times 3 = 9$ equivalence classes. A coarser partition leads to fewer iterations in the algorithms and ultimately shorter, more readable regular expressions output to the end user.

For both products we summarize our partition results in the following lemmas.

Lemma 13. *If* $\widehat{\mathit{to\_equivs}}_A$ *and* $\widehat{\mathit{to\_equivs}}_B$ *computes partitions and* $\widehat{\mathit{overlay}}_A$ *and* $\widehat{\mathit{overlay}}_B$ *preserves partitions then (a)* $\widehat{\mathit{to\_equivs}}_{A \times B}$ *computes a partition, (b)* $\widehat{\mathit{overlay}}_{A \times B}$ *preserves partitions, (c)* $\widehat{\mathit{to\_equivs}}_{A*B}$ *computes a partition, and (d)* $\widehat{\mathit{overlay}}_{A*B}$ *preserves partitions*

Lemma 14. *If* $\widehat{\mathit{to\_equivs}}_A$ *and* $\widehat{\mathit{overlay}}_A$ *and* $\widehat{\mathit{to\_equivs}}_B$ *and* $\widehat{\mathit{overlay}}_B$ *satisfy Properties 1, 2 then (a)* $\widehat{\mathit{to\_equivs}}_{A \times B}$ *and* $\widehat{\mathit{overlay}}_{A \times B}$ *also satisfy Properties 1, 2 and (b)* $\widehat{\mathit{to\_equivs}}_{A*B}$ *and* $\widehat{\mathit{overlay}}_{A*B}$ *satisfy Properties 1, 2*

For presentational purposes we have stated the results in terms of a Cartesian pair and a reduced/smashed pair, but the results hold for a general Cartesian product $\Pi_i A_i$ and for a general reduced/smashed product $\Pi_i (A_i \setminus \{\bot\}) \cup \{\bot\}$.

4 An example language and analysis

With the regular expression domain in place we are now in position to illustrate it with a static analysis. To this end we first study a concurrent, imperative programming language. Our starting point is a core imperative language structured into three syntactic categories of arithmetic expressions (e), Boolean expressions (b), and statements (s):

$$
\begin{aligned}
E \ni e &::= n \mid x \mid ? \mid e_1 + e_2 \mid e_1 - e_2 \\
B \ni b &::= \texttt{tt} \mid \texttt{ff} \mid x_1 < x_2 \\
S \ni s &::= \texttt{skip} \mid x := e \mid s\,;s \mid \texttt{if}\ b\ \texttt{then}\ s\ \texttt{else}\ s \mid \texttt{while}\ b\ \texttt{do}\ s\ \texttt{end} \\
&\quad \mid s \oplus s \mid ch?x \mid ch!e \mid \texttt{stop} \\
P \ni p &::= pid_1 : s_1 \parallel \ldots \parallel pid_n : s_n
\end{aligned}
$$

```
spawn server() {
   highscore = 0;
   while (true) {
      choose {
         { ask? cid
           hsc! highscore; }
       | { report? new
           if (highscore < new)
              { highscore = new; } } } } }
```

```
spawn client() {
   id = 0;
   best = 0;
   while (true) {
      ask! id;
      hsc? best;
      new = ?;
      if (best < new)
         { best = new;
           report! best; } } }
```

Fig. 9: A server and client sharing a high score

For presentational purposes we keep the arithmetic and Boolean expressions minimal. The slightly non-standard arithmetic expression '?' non-deterministically evaluates to *any* integer. The statements of the core language have been extended with primitives for non-deterministic choice ($\oplus$), for reading and writing messages from/to a named channel ($ch?x$ and $ch!e$), and for terminating a process (`stop`). The two message passing primitives are synchronous. To build systems of communicating processes, we extend the language further with a syntactic category of programs (p), consisting of a sequence of named processes.

As an example, consider a server communicating with a client as illustrated in Fig. 9. The server and the client each keep track of a 'highscore'. The client may query the server on the `ask`-channel and subsequently receive the server's current highscore on the `hsc`-channel. The client may also submit a new highscore to the server, using the `report`-channel. The example client performs an indefinite cycle consisting of a query followed by a subsequent response and a potential new highscore report. We can express this example as a program of our core process language.

We formulate in Fig. 10 a static analysis $\widehat{\mathcal{P}}$ which analyzes a process in isolation against an invariant for the context's communication. The analysis is formulated for a general abstract domain of values $\widehat{Val}$, e.g., intervals. To capture communication over a particular channel, we reuse an interval lattice (assuming the channels have been enumerated). This leads to a reduced product $Interval * \widehat{Val}$ for characterizing reads and an identical product for characterizing writes. We can then formulate a channel lattice for capturing both reads and writes: $\widehat{Ch}(\widehat{Val}) = (Interval * \widehat{Val}) \times (Interval * \widehat{Val})$. This product should not be reduced, as we do not wish to exclude processes that, e.g., only perform writes (with the read half of the channel domain being bottom). Finally we can plug the channel lattice into the regular expression domain: $\widehat{R}_{\widehat{Ch}(\widehat{Val})}$. In the static analysis in Fig. 10, f (for future) ranges over this domain. Intuitively, $\widehat{\rho}$ over-approximates the store (as traditional), whereas f over-approximates the signals of the environment (it is *consumed* by the analysis). The sequential analysis furthermore relies on an auxiliary function $\widehat{\mathcal{A}}$ for analyzing arithmetic expressions and two filter functions $\widehat{true}$ and $\widehat{false}$ to pick up additional information from variable comparisons (their definitions are available in the full version of this

$$
\begin{aligned}
\widehat{\mathcal{P}}[\![\texttt{skip}^\ell]\!] &= \lambda(\widehat{\rho}, f).(\widehat{\rho}, f)\\
\widehat{\mathcal{P}}[\![x :=^\ell e]\!] &= \lambda(\widehat{\rho}, f).(\widehat{assign}(\widehat{\rho}, x, \widehat{\mathcal{A}}(e, \widehat{\rho})), f)\\
\widehat{\mathcal{P}}[\![s_1 ;^\ell s_2]\!] &= \widehat{\mathcal{P}}[\![s_2]\!] \circ \widehat{\mathcal{P}}[\![s_1]\!]\\
\widehat{\mathcal{P}}[\![\texttt{if}^\ell\ b\ \texttt{then}\ s_1\ \texttt{else}\ s_2]\!] &= \lambda(\widehat{\rho}, f).\widehat{\mathcal{P}}[\![s_1]\!](\widehat{true}(b, \widehat{\rho}), f) \sqcup \widehat{\mathcal{P}}[\![s_2]\!](\widehat{false}(b, \widehat{\rho}), f)\\
\widehat{\mathcal{P}}[\![\texttt{while}^\ell\ b\ \texttt{do}\ s\ \texttt{end}]\!] &= \lambda(\widehat{\rho}, f).(\widehat{false}(b, \widehat{\rho}''), f'')\\
&\quad \text{where } (\widehat{\rho}'', f'') = \lim_i F^i(\widehat{\rho}, f) \text{ and } F(\widehat{\rho}', f') = (\widehat{\rho}', f') \triangledown \widehat{\mathcal{P}}[\![s]\!](\widehat{true}(b, \widehat{\rho}'), f')\\
\widehat{\mathcal{P}}[\![s_1 \oplus^\ell s_2]\!] &= \lambda(\widehat{\rho}, f).\widehat{\mathcal{P}}[\![s_1]\!](\widehat{\rho}, f) \sqcup \widehat{\mathcal{P}}[\![s_2]\!](\widehat{\rho}, f)\\
\widehat{\mathcal{P}}[\![ch?^\ell x]\!] &= \lambda(\widehat{\rho}, f).\ \bigsqcup_{\substack{[ch!\,v_a] \in \widehat{range}(f)\\ ch!\,v = \widehat{project}([ch!\,v_a])\\ \widehat{\mathcal{D}}_{\widehat{repr}([ch!\,v_a])}(f) \not\sqsubseteq \emptyset}} (\widehat{assign}(\widehat{\rho}, x, v), \widehat{\mathcal{D}}_{\widehat{repr}([ch!\,v_a])}(f))\\
\widehat{\mathcal{P}}[\![ch!^\ell e]\!] &= \lambda(\widehat{\rho}, f).\ \bigsqcup_{\substack{[ch?\,v_a] \in \widehat{overlay}(\widehat{range}(f), \widehat{to\_equivs}(ch?\,v'))\\ ch?\,v = \widehat{project}([ch?\,v_a])\\ v \sqcap v' \neq \bot\\ \widehat{\mathcal{D}}_{\widehat{repr}([ch?\,v_a])}(f) \not\sqsubseteq \emptyset}} (\widehat{\rho}, \widehat{\mathcal{D}}_{\widehat{repr}([ch?\,v_a])}(f)) \quad \text{where} \quad v' = \widehat{\mathcal{A}}(e, \widehat{\rho})\\
\widehat{\mathcal{P}}[\![\texttt{stop}^\ell]\!] &= \lambda(\widehat{\rho}, f).(\bot, f)
\end{aligned}
$$

Fig. 10: Analysis of the process language: $\widehat{\mathcal{P}} : S \longrightarrow \widehat{Store} \times \widehat{R}_{\widehat{Ch}(\widehat{Val})} \longrightarrow \widehat{Store} \times \widehat{R}_{\widehat{Ch}(\widehat{Val})}$

paper). Finally, the auxiliary function $\widehat{assign}$ defined as $\widehat{assign}(\widehat{\rho}, x, v) = \widehat{\rho}[x \mapsto v]$ models the effect of an assignment.

In the two cases for network read and write we utilize the shorthand notation $[ch!v_a]$ and $[ch?v_a]$ to denote equivalence classes $[\langle(\bot, \bot), ([ch; ch], [v_a; v_a])\rangle]$ and $[\langle([ch; ch], [v_a; v_a]), (\bot, \bot)\rangle]$ over atom writes and atom reads in $\widehat{Ch}(\widehat{Val})$, respectively. Both of these cases utilize the Brzozowski derivative $\widehat{\mathcal{D}}$ of f to anticipate all possible writes and reads from the network environment. For example, if we analyze a read statement $\texttt{in?x}$ in an abstract store $\widehat{\rho}$ and in a network environment described by $\texttt{in!}[1; 1000] \cdot r$ (for some $r \not\sqsubseteq \emptyset$) we first assume channel names have been numbered, e.g., mapping channel name '$\texttt{in}$' to 0. For readability, we therefore write $\texttt{in!}[1; 1000] \cdot r$ instead of $\langle(\bot, \bot), ([0; 0], [1; 1000])\rangle \cdot r$ where the channel name $\texttt{in}$ should be understood as 0 (which we can capture precisely with the intervals as $[0; 0]$) and where we similarly utilize the above shorthand notation. Now $\widehat{range}(\texttt{in!}[1; 1000] \cdot r) = \widehat{range}(\texttt{in!}[1; 1000]) = \widehat{to\_equivs}(\texttt{in!}[1; 1000])$ returns a partition that includes the equivalence class $[\texttt{in!}[1; 1000]]$. Furthermore $\widehat{project}([\texttt{in!}[1; 1000]]) = \texttt{in!}[1; 1000]$ and $\widehat{repr}([\texttt{in!}[1; 1000]])$ returns an atom in this equivalence class, e.g., $\texttt{in!}[1; 1]$ such that $\widehat{\mathcal{D}}_{\texttt{in!}[1;1]}([\texttt{in!}[1; 1000]] \cdot r) = \epsilon \cdot r =_{ACI+} r \not\sqsubseteq \emptyset$. The analysis therefore includes $(\widehat{assign}(\widehat{\rho}, \texttt{x}, [1; 1000]), r) = (\widehat{\rho}[\texttt{x} \mapsto [1; 1000]], r)$ as an approximate post-condition for the read statement. When the analysis attempts to derive wrt. atoms from other equivalence classes, e.g., the atom $\texttt{in!}[1001; 1001]$ we get $\widehat{\mathcal{D}}_{\texttt{in!}[1001;1001]}([0![1; 1000]] \cdot r) = \emptyset \cdot r =_{ACI+}$

$\emptyset$ and such contributions are therefore disregarded. As usual, the analysis can incur some information loss, e.g., if each branch of a conditional statement contains a read into the same variable. These values will then be over-approximated by the join of the underlying value domain.

For the interval domain of values, we stick to the traditional widening operator [7]. For the abstract stores, we perform a traditional pointwise lift $\dot{\triangledown}$ of the interval widening for each store entry. For regular expressions, the situation is more interesting: In the search for a while-loop invariant, new futures can only appear as derivatives of the loop's initial future. Since there are only a finite number of these up to ACI of $+$, an upward Kleene iteration is bounded and hence does not require widening. The resulting widening operator over analysis pairs can therefore be expressed as follows: $(\widehat{\rho}_1, f_1) \triangledown (\widehat{\rho}_2, f_2) = (\widehat{\rho}_1 \mathbin{\dot{\triangledown}} \widehat{\rho}_2, f_1 + f_2)$.

For example, under the worst-case assumption of *any* context communication ($\top^*$), the analysis will determine the following server invariant for the highscore example, expressed as an abstract store and a regular expression over channel-labeled intervals: $[\,\texttt{cid} \mapsto [-\infty; +\infty];\ \texttt{highscore} \mapsto [0; +\infty];\ \texttt{new} \mapsto [-\infty; +\infty]\,]$ and $\top^*$. When analyzed under the erroneous policy of receiving (non-negative payload) messages in the wrong order $(\texttt{ask!}[0; +\infty] + \texttt{report!}[0; +\infty] \cdot \texttt{hsc?}[-\infty; +\infty])^*$ the analysis infers the following stronger invariant for the server: $[\,\texttt{highscore} \mapsto [0; +\infty];\ \texttt{new} \mapsto [0; +\infty]\,]$ and $(\texttt{ask!}[0; +\infty] + \texttt{report!}[0; +\infty] \cdot \texttt{hsc?}[-\infty; +\infty])^*$ and that `hsc! highscore` in line number 6 cannot execute successfully.

5 Implementation

We have implemented a prototype of the analysis in OCaml. Currently the prototype spans approximately 5000 lines of code. Each lattice (intervals, abstract stores, ...) is implemented as a separate module, with suitable parameterization using functors, e.g., for the generic regular expression domain. The partition of lattice atoms is implemented by requiring that a parameter lattice A implements $\widehat{\mathit{to\_equivs}}$ and $\widehat{\mathit{overlay}}$ with signatures as listed in Sec. 3. To gain confidence in the implementation, we have furthermore performed randomized, property-based testing (also known as 'quickchecking') of the prototype. The QuickCheck code takes an additional ~ 650 lines of code. We quickchecked the individual lattices for typical lattice properties (partial order properties, associativity and commutivity of join and meet, etc.) and the lattice operations ($\widehat{\mathcal{D}}$, $\cdot$, etc.) for monotonicity, using the approach of Midtgaard and Møller [24]. This approach was fruitful in designing and testing the suggested ordering algorithm (and its implementation) and in our implementations of $\widehat{\mathit{to\_equivs}}$ and $\widehat{\mathit{overlay}}$. To increase our confidence in the suggested widening operator, we furthermore extended the domain-specific language of Midtgaard and Møller [24] with the ability to test whether lattice-functions are increasing, when applied to arbitrarily generated input. We then used this ability to test all involved widening operators. QuickCheck immediately found a bug in an earlier version of our widening algorithm, which was not increasing in the second argument on the input $(\epsilon, (\top^* \cdot \mathit{odd} \cdot \mathit{odd}) \mathbin{\&} \complement\, \epsilon)$ (here again listed over the parity domain). The

corresponding automaton computed for this counterexample turns out to have a strongly connected component, which led us to find and patch an early erroneous attempt to identify and remove explicit error states.

With the domain and analysis implemented and tested, we can apply it to the example program from Sec. 4 and we obtain the reported results. We have also analyzed a number of additional example programs, including several from the literature: two CSP examples from Cousot and Cousot [8], a simple math server adapted from Vasconcelos, Gay, and Ravara [28], and a simple authentication protocol from Zafiropulo et al. [31]. For each of these examples, the analysis prototype completes in less than 0.003 seconds on a lightly loaded laptop with a 2.8 Ghz Intel Core i5 processor and 8 GB RAM. While this evaluation is encouraging it is also preliminary. We leave a proper empirical evaluation of the approach for future work. The source code of the prototype, the corresponding QuickCheck code, and our examples are available as downloadable artifacts.[5] Our proofs are available in the full version of this paper.[6]

6 Related work

Initially Cousot and Cousot developed a static analysis for Hoare's Communicating Sequential Processes (CSP) [8]. Our example analysis also works for a CSP-like language, but differs in the means to capture communication, where we have opted for lattice-valued regular expressions. A line of work has since developed static analyses for predicting the *communication topology* of mobile calculi. For example, Venet [30] developed a static analysis framework for π-calculus and Rydhof Hansen et al. [16] develop a control-flow analysis and an occurrence counting analysis for mobile ambients. Whereas the communication topology is apparent from the program text of our process programs, we instead focus on analyzing the *order* and the *content* of such communication by means of lattice-valued regular expressions.

Historically, the Communicating Finite State Machines (CFSMs) [5] have been used to model and analyze properties of protocols. CFSMs express a distributed computation as a set of finite state automata that communicate via (buffered) message passing over channels. We refer to Le Gall, Jeannet, and Jéron [13] for an overview of (semi-)algorithms and decidability results within CFSMs. Le Gall, Jeannet, and Jéron [13] themselves developed a static analysis for analyzing the communication patterns of FIFO-queue models in CFSMs. In a follow-up paper, Le Gall and Jeannet [12] developed the abstract domain of *lattice automata* (parameterized by an atomic value lattice), thereby lifting a previous restriction to finite lattices. Our work differs from Le Gall and Jeannet's in that it starts from the language-centric, lattice-valued regular expressions, as opposed to the decision-centric, lattice-valued finite automata (one can however translate one formalism to the other). The two developments share a common de-

[5] `https://github.com/jmid/regexpanalyser`
[6] `http://janmidtgaard.dk/papers/Midtgaard-Nielson-Nielson:SAS16-full.pdf`

pendency on atomistic lattices:[7] Lattice automata require atoms (and partitions over these) as its labels, whereas our co-inductive ordering algorithm relies on Brzozowski derivatives wrt. atoms (and partitions over these). We see advantages in building on Brzozowski derivatives: (a) we can succinctly express both intersection (meet) and complement symbolically in the domain, (b) we immediately inherit a "one-step normal form" from the underlying equation (Theorem 1), whereas Le Gall and Jeannet develop a class of 'normalized lattice automata', and (c) our ordering algorithm lazily explores the potentially exponential space of derivatives (states) and bails early upon discovering a mismatch.

Our work has parallels to previous work by Lesens, Halbwachs, and Raymond (LHR) on inferring *network invariants* for a linear network of synchronously communicating processes [20]. Similar to us, they use a regular language to capture network communication. They furthermore allow *network observers* to monitor network communication and emit disjoint *alarms* if a desired property is not satisfied. They primarily consider a greatest fixed point expressing satisfiability of a desired network invariant, which they under-approximate by an analysis over a regular domain using a *dual widening* operator, that starts above and finishes below the greatest fixed point. Our work differs in that LHR abstract away from the concrete syntax of processes whereas we instead attempt to lift existing analysis approaches. As a consequence, LHR target a fixed communication topology, whereas in our case, which process that reads another process' output depend on their inner workings. As pointed out by LHR, widening operators have to balance convergence speed and precision. They discuss possible design choices and settle on a (dual) operator that makes an extreme tradeoff, by being very precise but sacrificing guaranteed convergence. On the contrary we opt for a convergence guarantee at the cost of precision. On the other hand, their *delayed widening* technique to further improve precision, is likely to also improve our present widening further. Whereas our widening operator is less precise, we believe LHR's automata with powersets of signals fits immediately our atomistic Galois insertion condition. Finally LHR's approach depends on determinising automata which incurs a worst case exponential blow up. They therefore seek to avoid such determinization in future work. Since lattice-valued regular expressions require less determinization (writing out the equations in steps 2,3 of Fig. 6 before collapsing them in step 5 requires determinization), they represent a step in that direction.

Our process analysis approach is inspired by an approach of Logozzo [21] for analyzing classes of object-oriented programs. Logozzo devises a modular analysis of class invariants using contexts approximated by a lattice-valued regular expression domain to capture calling policies. Like us, Logozzo builds on a language inclusion ordering but he does not develop an algorithm for computing it. Before developing the current widening operator we experimented with his structural widening operator based on symbolic pattern matching of two given regular expressions. QuickCheck found an issue with the definition: The original

[7] These are however referred to as 'atomic lattices' contradicting standard terminology [15, 10].

definition [21, Fig.3] allows $(odd \cdot even) \bigtriangledown_r odd = (odd \bigtriangledown_r odd) \cdot even = odd \cdot even$ which is not partially ordered with respect to its second argument under a language inclusion ordering.

Owens, Reppy, and Turon [25] report on using derivatives over extended regular expressions (EREs) for building a scanner generator. In doing so, they revisit Brzozowski's original constructions in a functional programming context. To handle large alphabets such as Unicode, they extend EREs (conservatively) with *character sets*, allowing a subset of characters of the input alphabet as letters of their regular expressions. From our point of view, the extension can be seen as EREs with characters over a powerset lattice. Overall their experiments show that a well-engineered scanner generator will explore only a fraction of all possible derivatives, and in many cases compute the minimal automaton. Our implementation is inspired by that of Owens, Reppy, and Turon [25], in that it uses (a) an internal syntax tree representation that maps ACI-equivalent regular expressions to the same structure and (b) an interface of *smart constructors*, e.g., to perform simplifying reductions.

A line of work has concerned axiomatizing equivalence (and containment) of regular expressions (REs) and of the more general Kleene algebras [27, 19, 14, 17]. We refer to Henglein and Nielsen [17] for a historical account of such developments. Grabmayer [14] gave a co-inductive axiomatization of RE equivalence based on Brzozowski derivatives and connects it to an earlier axiomatization of Salomaa [27]. In particular, our *nullable* function corresponds to Grabmayer's o-function, and his COMP/FIX proof system rule concludes that two REs E and F are equivalent if $o(E) = o(F)$ and if all derivatives $\mathcal{D}_a(E) = \mathcal{D}_a(F)$ are equivalent much like our co-inductive `leq_test` for deciding containment checks for *nullable* and queries all derivatives for containment. In fact, we can turn Fig. 5 into an equivalence algorithm akin to Grabmayer [14] by simply replacing the implication in line number 6 with `if` $nullable(r_1) \iff nullable(r_2)$. Kozen's axiomatization of Kleene algebras and his RE completeness proof of these [19] have a number of parallels to the current work: (a) the axiomatization contains a conditional inclusion axiom similar to Arden's lemma, (b) our k-limited partitioning $\approx_k^{col_r}$ can be viewed as an approximation of Kozen's *Myhill-Nerode equivalence relation* that algebraically expresses state minimization, and (c) the completeness proof involves solving matrices over REs (which themselves form a Kleene algebra) in a manner reminiscent of Brzozowski's translation scheme. To synthesize a regular expression from the collapsed equations we could alternatively have used Kozen's approach that partitions the matrix into sub-matrices with square sub-matrices on the diagonal and recursively solves these. Henglein and Nielsen [17] themselves gave a co-inductive axiomatization of RE containment, building on strong connections to type inhabitation and sub-typing.

Our work also has parallels to Concurrent Kleene Algebra (CKA) [18]. In particular, CKA is based on a set-of-traces ordering—a language inclusion ordering—in which a set of possible traces describes program event histories, akin to our example analysis. Furthermore, CKA's extension over Kleene algebra to include a parallelism operator could be a viable path forward to extend the proposed

example analysis from a single process in a network environment to support arbitrary process combinations.

Within model checking over timed automata [1], there are parallels between partitioning clock interpretations over timed transition tables into regions and our partitioning of lattice atoms into equivalence classes. The two developments however differ in that timed Büchi and Muller automata naturally target liveness properties (by their ability to recognize ω-regular languages), whereas we for now target safety properties with lattice-valued regular expressions. The extension to parametric timed automata [2] allow for enriching the expressible relations on transitions. In the current framework this would correspond to instantiating the lattice-valued regular expressions with a relational abstract domain. In future work we would like to investigate the degree to which such instantiations are possible.

7 Conclusion

We have developed lattice-valued regular expressions as an abstract domain for static analysis including a co-inductive ordering algorithm and a widening operator. As an illustration of the parametric domain we have presented a static analysis of communication properties of a message-passing process program against a given network communication policy. Lattice-valued regular expressions constitute an intuitive and well-known formalism for expressing such policies. We plan to reuse the domain for further message-passing analysis in the future.

References

[1] R. Alur and D. L. Dill. A theory of timed automata. *TCS*, 126(2):183–235, 1994.
[2] R. Alur, T. A. Henzinger, and M. Y. Vardi. Parametric real-time reasoning. In *STOC'93*, pages 592–601, 1993.
[3] D. N. Arden. Delayed-logic and finite-state machines. In *2nd Annual Symposium on Switching Circuit Theory and Logical Design*, pages 133–151. IEEE Computer Society, 1961.
[4] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *PLDI'93*, pages 46–55, 1993.
[5] D. Brand and P. Zafiropulo. On communicating finite state machines. *JACM*, 30:323–342, 1983.
[6] J. A. Brzozowski. Derivatives of regular expressions. *JACM*, 11(4):481–494, 1964.
[7] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *ISOP'76*, pages 106–130. Dunod, Paris, France, 1976.
[8] P. Cousot and R. Cousot. Semantic analysis of Communicating Sequential Processes. In *ICALP'80*, volume 85 of *LNCS*, pages 119–133, 1980.
[9] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.
[10] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, second edition, 2002.
[11] J. Feret. Abstract interpretation-based static analysis of mobile ambients. In *SAS'01*, volume 2126 of *LNCS*, pages 412–430, 2001.

[12] T. L. Gall and B. Jeannet. Lattice automata: A representation for languages on infinite alphabets, and some applications to verification. In *SAS'07*, volume 4634 of *LNCS*, pages 52–68, 2007.
[13] T. L. Gall, B. Jeannet, and T. Jéron. Verification of communication protocols using abstract interpretation of FIFO queues. In *AMAST'06*, volume 4019 of *LNCS*, pages 204–219, 2006.
[14] C. Grabmayer. Using proofs by coinduction to find "traditional" proofs. In *CALCO'05*, pages 175–193, 2005.
[15] G. Grätzer. *General Lattice Theory*. Academic Press, 1978.
[16] R. R. Hansen, J. G. Jensen, F. Nielson, and H. R. Nielson. Abstract interpretation of mobile ambients. In *SAS'99*, volume 1694 of *LNCS*, pages 134–148, 1999.
[17] F. Henglein and L. Nielsen. Regular expression containment: Coinductive axiomatization and computational interpretation. In *POPL'11*, pages 385–398, 2011.
[18] T. Hoare, S. van Staden, B. Möller, G. Struth, J. Villard, H. Zhu, and P. O'Hearn. Developments in concurrent Kleene algebra. In *RAMiCS 2014*, volume 8428 of *LNCS*, pages 1–18, 2014.
[19] D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994.
[20] D. Lesens, N. Halbwachs, and P. Raymond. Automatic verification of parameterized linear networks of processes. In *POPL'97*, pages 346–357, 1997.
[21] F. Logozzo. Separate compositional analysis of class-based object-oriented languages. In *AMAST'04*, volume 3116 of *LNCS*, pages 334–348, 2004.
[22] J. C. Martin. *Introduction to Languages and the Theory of Computation*. McGraw-Hill, 1997.
[23] L. Mauborgne. Tree schemata and fair termination. In *SAS'00*, volume 1824 of *LNCS*, pages 302–321, 2000.
[24] J. Midtgaard and A. Møller. Quickchecking static analysis properties. In *ICST'15*, pages 1–10. IEEE Computer Society, 2015.
[25] S. Owens, J. Reppy, and A. Turon. Regular-expression derivatives re-examined. *Journal of Functional Programming*, 19(2):173–190, 2009.
[26] G. Rosu and M. Viswanathan. Testing extended regular language membership incrementally by rewriting. In *RTA'03*, volume 2706 of *LNCS*, pages 499–514, 2003.
[27] A. Salomaa. Two complete axiom systems for the algebra of regular events. *JACM*, 13(1):158–169, 1966.
[28] V. T. Vasconcelos, S. Gay, and A. Ravara. Typechecking a multithreaded functional language with session types. *TCS*, 368(1–2):64–87, 2006.
[29] A. Venet. Abstract cofibered domains: Application to the alias analysis of untyped programs. In *SAS'96*, volume 1145 of *LNCS*, pages 366–382, 1996.
[30] A. Venet. Automatic determination of communication topologies in mobile systems. In *SAS'98*, volume 1503 of *LNCS*, pages 152–167, 1998.
[31] P. Zafiropulo, C. H. West, H. Rudin, D. D. Cowan, and D. Brand. Towards analyzing and synthesizing protocols. *IEEE Transactions on Communications*, Com-28(4):651–661, 1980.